An idea: SAM Access Through the Familiar File System

Adam Lyon CD/REX

March 2011

A barrier to adopting SAM is that data lookup and access are very different than a user's normal experience. Nearly all users discover and access files through the unix file system. That is using `ls` to search directories for the files they want and then opening those files with their application by supplying a file system path.

Because SAM has a very rich set of metadata, file discovery is done with database queries (with a complicated `sam translate constraints` wrapper as the interface for the user). Files are then retrieved out of SAM by creating a dataset definition and then issuing SAM commands to copy the files to a local area or cache. The concepts here and commands are difficult to use and there is a steep learning curve for new users. This situation is seen as a large barrier to experiments adopting SAM, especially if they are used to having their files in a regular file system (e.g. Bluearc).

THE IDEA: Can we provide a limited, but useful, interface to SAM by emulating a familiar file system so that unix commands like `ls`, `cat`, `cp`, etc work? Can we present a filesystem with directories that correspond to particular database queries and then have a way to bring files into a local cache when they are accessed via this filesystem?

ENABLING TECHNOLOGY: There is a linux package called FUSE (http://fuse.sourceforge.net/) (works on the Mac too) that implements a fully functional filesystem interface to a user program. FUSE has been around for many years and has been quite stable - it is not new technology. In a nutshell, you write a library that reacts to filesystem calls (e.g. read directory, get file attributes, open file, read file, follow symlink, remove file). You can define file, directory, symlink, etc as interface items in your program - they can be whatever you want. A popular use of FUSE is `sshfs` (I use this often on my mac). With `sshfs` you can access files on a remote node as if they were local (e.g. you can directly load a remote file into Emacs as if it were sitting on your machine -- but in fact the remote file is read and updated). For example, I can access a file on my clued0 node from my laptop by using a path like `/mnt/adam-clued0-node/home/lyon/myfile.txt`. FUSE is used to intercept local file system calls and issue the relevant `ssh` commands to handle the remote filesystem. It is very convenient.

A very nice feature of FUSE is that there are bindings to the C++ code for many languages. There are in fact three separate bindings for python. Having a python binding is especially convenient as we also have a python interface to SAM. That makes interacting with SAM quite easy, and of course programming in python has its usual advantages. Note that the FUSE code can intercept read and write calls, and they can be handled by the python layer. So this can mean that if an application wants to read the file, python may be responsible for streaming out the file; a task better left to the operating system for efficiency. A mitigation is to make "files" symbolic links. FUSE will intercept the call to

determine the symbolic link target. The python code can supply a path to the real file (e.g. in some cache), and then the OS will happily go about feeding it to the application without python being involved at all.

A CRUDE PROOF OF PRINCIPLE `samfs`: On a fermicloud virtual machine, I have implemented a very crude example filesystem using FUSE that connects to SAM at D0. It is written using the `fuse24.py` python binding. With this `samfs`, a user can look up data files corresponding to runs and data-tiers. A user can also look up files satisfying a dataset definition. If a file is accessed, it is "copied" to a cache area (the user never needs to know where this area is). In fact this latter function is simulated and the files are filled with fake contents.

Let's take a little tour of this crude `samfs`. When doing directories, I'll use `ls -F`. The `-F` appends names with a symbol indicating the type. `/` means a directory, `*` is a regular file and `@` is a symbolic link.

A main use case that I'm trying to handle is interactive use where you want to get a file and play with it to, say, test your code. This is a use case that many users are concerned about, because the regular SAM interface requires the learning curve.

```
<fermicloud052> ls -F /samfs
dataset/  get/  README.txt@  run/  store/
```

First, you see that this fake filesystem is mounted at `/samfs`. None of the items in this directory are real - they are all simply strings returned by the python code that are interpreted by the OS as file system objects. There's some help here in the `README.txt` file. `README.txt` is not a real file. The program intercepts calls to it and tells the OS that it is a symbolic link pointing to a real text file in the program area. So when you `cat` that file, FUSE intercepts the call and returns the target for the fake symbolic link and then the OS opens and displays the file.

```
<fermicloud052> cat /samfs/README.txt
SAMFS - SAM through the File System

Files by runs:
ls /samfs/run/268211  # GIves data_tiers as directories
ls /samfs/run/268211/raw  # GIves raw files for this run

Files in a dataset
ls /samfs/dataset/lyon  # Gives all of lyon's datasets as directories
ls /samfs/dataset/lyon/lyonWZEM1305_prk993l1  # Gives files in this dataset

Files that are symbolic links are in the cache and can be opened
directly.

Files that are not symbolic links are not in the cache. Use the "get"
path to retrieve them. E.g.

cat /samfs/get/dataset/lyon/lyonWZEM1305_prk993l1/myfile.root
```

```
This will put the file in the cache and then will open it.

Wildcards with a get path is not allowed. You cannot do a directory
listing with a get path (since that will inadvertently retrieve all of
the files)

You can store files as well into SAM with this interface.
You must copy your data file and the metadata file into the SAM Store
holding area (where ever that is). Then copy the data file into
/samfs/store (the source path can be the original place of the file or
the location in the storage area [/samfs/store is not a real
location]).

e.g.
cp myFile.root /sam/store1
cp myFile.py /sam/store1  # metadata
cp myFile.root /samfs/store

You can see progress with
cat /samfs/store/myFile.root

You can watch progress with
watch -n 2 'cat /samfs/store/myFile.root'
```

Let's look at files associated with runs.

```
<fermicloud052> ls -F /samfs/run
ERROR-Supply-a-run-number-like-268211@
```

Since there are 1000s of run numbers at D0, it makes no sense to list them all. Instead, a "file" is displayed to indicate an error telling the user to give a run number.

```
<fermicloud052> ls -F /samfs/run/268211
raw/  root-tree/  root-tree-bygroup/  thumbnail/  thumbnail-bygroup/
```

Within the "run" directory are sub-directories of data-tiers (again, these are not real directories but are strings returned by the python code). Let's look at raw data.

```
<fermicloud052> ls -F /samfs/run/268211/raw
all_1_0000268211_004.raw*  all_2_0000268211_003.raw*  all_3_0000268211_018.raw*
all_1_0000268211_012.raw*  all_2_0000268211_004.raw*  all_3_0000268211_021.raw*
all_1_0000268211_016.raw*  all_2_0000268211_008.raw*  all_3_0000268211_027.raw*
all_1_0000268211_020.raw*  all_2_0000268211_020.raw@  all_3_0000268211_030.raw*
```

```
all_1_0000268211_027.raw*  all_2_0000268211_030.raw*  all_4_0000268211_009.raw*
all_1_0000268211_031.raw*  all_2_0000268211_036.raw*  monitor_2_0000268211_003.raw*
all_1_0000268211_033.raw*  all_3_0000268211_008.raw*  monitor_4_0000268211_002.raw*
all_1_0000268211_034.raw*  all_3_0000268211_012.raw*
all_1_0000268211_035.raw*  all_3_0000268211_014.raw*
```

To determine the list of files, the python code calls:

```
 sam translate constraints --dim='run_number @run@ and data_tier @data_tier@' --nosummary
```

where the items in @ are supplied by the program (note that I could call the SAM python API directly, but it was easier to call the shell. Clearly this is not an efficient way to do this).

Also note that the number of actual files is very large (100s of files). I have an artificial cut off of 25 files to keep the output small.

Note that I can do the regular file system like stuff with wildcards... (this comes for free - I don't have to program any of this in the python code)

```
<fermicloud052> ls -F /samfs/run/268211/raw/monitor*
/samfs/run/268211/raw/monitor_2_0000268211_003.raw*
/samfs/run/268211/raw/monitor_4_0000268211_002.raw*
<fermicloud052>
<fermicloud052> ls -F /samfs/run/268211/raw/all_3_*.raw
/samfs/run/268211/raw/all_3_0000268211_008.raw*  /samfs/run/268211/raw/all_3_0000268211_021.raw*
/samfs/run/268211/raw/all_3_0000268211_012.raw*  /samfs/run/268211/raw/all_3_0000268211_027.raw*
/samfs/run/268211/raw/all_3_0000268211_014.raw*  /samfs/run/268211/raw/all_3_0000268211_030.raw*
/samfs/run/268211/raw/all_3_0000268211_018.raw*
<fermicloud052>
<fermicloud052> ls -F /samfs/run/268211/raw/*003.raw
/samfs/run/268211/raw/all_2_0000268211_003.raw*  /samfs/run/268211/raw/monitor_2_0000268211_003.raw*
```

Note that all of the files have a *, meaning they are not cached. If I try to access one, I get...

```
<fermicloud052> cat /samfs/run/268211/raw/all_2_0000268211_020.raw
cat: /samfs/run/268211/raw/all_2_0000268211_020.raw: No such file or directory
```

Because retrieving the file is an extremely expensive operation, you don't want to do it lightly. FUSE can't tell the difference between dereferencing a symbolic link for `ls` and for reading the file. So without protection, one could easily inadvertently move a large number of files out of SAM by simply doing an `ls`. We don't want that. So to move a file the user has to do something special (but hopefully simple). My idea is to add a `get` to the path. The code knows not to allow `ls` if `get` is in the path (so you don't try to move every file in the "directory").

```
<fermicloud052> cat /samfs/get/run/268211/raw/all_2_0000268211_020.raw
This is file all_2_0000268211_020.raw
The end
```

This use of `/samfs/get/...` moves the file into the local cache. `cat` displays it (the code creates the file and puts in it fake data, but in reality a `sam get dataset --fileName=@filename@` would occur here).

Now doing an `ls` (without `get`) shows the file is cached...

```
<fermicloud052> ls -F /samfs/run/268211/raw
all_1_0000268211_004.raw*  all_2_0000268211_003.raw*  all_3_0000268211_018.raw*
all_1_0000268211_012.raw*  all_2_0000268211_004.raw*  all_3_0000268211_021.raw*
all_1_0000268211_016.raw*  all_2_0000268211_008.raw*  all_3_0000268211_027.raw*
all_1_0000268211_020.raw*  all_2_0000268211_020.raw@  all_3_0000268211_030.raw*
all_1_0000268211_027.raw*  all_2_0000268211_030.raw*  all_4_0000268211_009.raw*
all_1_0000268211_031.raw*  all_2_0000268211_036.raw*  monitor_2_0000268211_003.raw*
all_1_0000268211_033.raw*  all_3_0000268211_008.raw*  monitor_4_0000268211_002.raw*
all_1_0000268211_034.raw*  all_3_0000268211_012.raw*
all_1_0000268211_035.raw*  all_3_0000268211_014.raw*
```

Note the @ after our cached file. You can read it with that path, or use the `get` path again (it checks if the file is already cached and won't do it again). In reality, you can use any application. For example, in Root you could do:

```
root [1]: TFile* myFile = new TFile("/samfs/get/run/268211/all_2_0000268211_020.raw")
```

The beauty of using `get` is that if the file is not cached, it will get it. The open will just take (perhaps a lot) longer. But operationally, you do what you are used to and it is easy. For example (first path does not have `get`, second does)...

```
<fermicloud052> cat /samfs/run/268211/raw/all_2_0000268211_020.raw
This is file all_2_0000268211_020.raw
```

```
The end
<fermicloud052>
<fermicloud052> cat /samfs/get/run/268211/raw/all_2_0000268211_020.raw
This is file all_2_0000268211_020.raw
The end
```

Note I tried to put in protection in case of making mistakes with `get`. You cannot do `ls` if `get` is in the path. Wildcards are not allowed.

```
<fermicloud052> # Without protection, this would retrieve every file (yikes!)
<fermicloud052> ls /samfs/get/run/268211/raw/
ERROR-Trying-to-list-directory-with-a-GET-path
<fermicloud052>
<fermicloud052> # Wildcards could also cause a large retrieval
<fermicloud052> cat /samfs/get/run/268211/raw/all*
cat: /samfs/get/run/268211/raw/all*: No such file or directory
```

Unfortunately, the wildcard checking occurs at the `get attribute` step and so it is not possible to return a nice error message. This is a deficiency of FUSE discussed later.

Let's look at the dataset queries.

```
<fermicloud052> ls /samfs/dataset
ERROR-Supply-a-user-name-like-lyon
```

There are lots of users, so it makes no sense to list them all. Supply one.

```
<fermicloud052> ls -F /samfs/dataset/lyon
166188-166189-raw/             lyonWZEM1306_prk670l1/
200203_streamtest_thumbnails/  lyonWZEM1306_prk784l1/
adamwze1304/                   run174805-onetmb/
adamwzmu1304/                  special-atg-streamingtest/
lyon_test_20030123/            specialstream_stream2thmb_20020103/
lyon_test_20030123b/           specialstream_stream2thmb_20020103a/
lyonWZEM1305_prk741l1/         streamtest-174806-tmb/
lyonWZEM1305_prk745l1/         streamtest-174806-tmb-1miss/
```

```
lyonWZEM1305_prk748l1/          streamtest_20021015_root/
lyonWZEM1305_prk763l1/          streamtest_20021122_raw/
lyonWZEM1305_prk773l1/          streamtest_200303_thumbnails/
lyonWZEM1305_prk784l1/          test20030505/
lyonWZEM1305_prk993l1/
```

There are the datasets that I have created (I artificially cut off at the first 25 as mentioned above). Notice they are all directories since they contain files. I get this list from:

```
 sam list definitions --userName=@username@
```

Let's look at a dataset definition...

```
<fermicloud052> ls -F /samfs/dataset/lyon/lyonWZEM1305_prk741l1
WZskim-emStream-20021221-112018.raw_p13.05.00@  WZskim-emStream-20021221-185419.raw_p13.05.00*
WZskim-emStream-20021221-114027.raw_p13.05.00*  WZskim-emStream-20021221-191841.raw_p13.05.00*
WZskim-emStream-20021221-115845.raw_p13.05.00*  WZskim-emStream-20021221-193921.raw_p13.05.00*
WZskim-emStream-20021221-183224.raw_p13.05.00*  WZskim-emStream-20021221-233546.raw_p13.05.00*
```

To get the list of files that satisfy the dataset definition, he python code is actually calling:

```
sam translate constraints --dim="__set__ @defName@" --nosummary
```

Like with the run data, I can bring these files into the cache and `ls` marks the file accordingly.

```
<fermicloud052> cat /samfs/get/dataset/lyon/lyonWZEM1305_prk741l1/WZskim-emStream-20021221-112018.raw_p1
3.05.00
This is file WZskim-emStream-20021221-112018.raw_p13.05.00
The end
<fermicloud052>
<fermicloud052> ls -F /samfs/dataset/lyon/lyonWZEM1305_prk741l1
WZskim-emStream-20021221-112018.raw_p13.05.00@  WZskim-emStream-20021221-185419.raw_p13.05.00*
WZskim-emStream-20021221-114027.raw_p13.05.00*  WZskim-emStream-20021221-191841.raw_p13.05.00*
WZskim-emStream-20021221-115845.raw_p13.05.00*  WZskim-emStream-20021221-193921.raw_p13.05.00*
WZskim-emStream-20021221-183224.raw_p13.05.00*  WZskim-emStream-20021221-233546.raw_p13.05.00*
```

Another topic - storing files into SAM is typically not an easy operation for users. `samfs` can improve things here. The difficulty is that SAM must know about two files - the main data file and the metadata file. For this to work, the user must copy both such files into some SAM

upload directory (not in `samfs`). But then one can trigger the `sam store` with another `cp` into `/samfs/store`, which is much easier than before (though really, the metadata is what makes this process difficult). Checking the status of the store is not easy either. This situation can be improved by having a fake file in `/samfs/store` that can contain the status information. For example (no real SAM store happens in the python code), ...

```
<fermicloud052> cd ~/samUpload  # A fake upload area
<fermicloud052> ls # my files to upload (the py file is the metadata file)
myDataFile.py  myDataFile.root
<fermicloud052>
<fermicloud052> ## Trigger the SAM store
<fermicloud052> cp myDataFile.root /samfs/store
<fermicloud052>
<fermicloud052> ## Check the status
<fermicloud052> ls /samfs/store
myDataFile.root
<fermicloud052>
<fermicloud052> cat /samfs/store/myDataFile.root
Store for myDataFile.root is in progress
```

OTHER USES: This crude proof of principle has three functions: locating and retrieving data via runs, locating and retrieving data via dataset definitions, and storing files into SAM. All of these functions are geared towards interactive use. One can imagine other functions as well. Marc Mengel had an interesting idea for batch jobs, instead of issuing `sam get next file` the application could try to open `/samfs/project/@sam_project@/@processId@/nextFile`. A complication is that SAM needs to know when the file is released by the application. Touching `/samfs/project/@sam_project@/@processId@/release` may be a solution.

Another interesting idea from Marc is to have `/samfs` fronted by a BestMan SRM Service. Since `/samfs` *is* a file system, one can imagine projecting SAM services with BestMAN. File `ls` requests and retrieval requests could occur with `srm` commands on the remote side, and then `samfs` FUSE is called by BestMAN (BestMAN just handles it like any other file system) to retrieve the information (or the files themselves) and send them to the remote user. This mechanism would mean that remote sites would not need a SAM client.

PROBLEMS WITH FUSE: 1) FUSE needs a kernel extension in order to work. The SLF5 distribution includes this extension, but the admins need to activate it (I think this is a one-time process). Installation on SLF4 machines may be difficult (but maybe that doesn't matter much anymore). 2) Because the filesystem is an OS element, the error messages one gets are limited to what is allowed by the OS. Sometimes, when doing an `ls` a fake file can be returned that has error information in the name and that can be descriptive. But otherwise one is limited to OS errors which may be hard to interpret. E.g. It may be hard to learn why a "No such file or directory" error is given. 3) There must be protection so that large number of files are not inadvertently retrieved from SAM. My crude code has such protection, but that leads to some inconveniences (the `get` path requirement). 4) Some SAM operations do not lend themselves to the filesystem metaphore. Care should be taken

so that `samfs` does not get too weird.

CONCLUSION: I think something like `samfs` can be used to give users an interface they are already familiar with. Coupled with BestMAN, `samfs` may make it easier to deploy SAM on remote sites as a client may not be needed there.